A Controlled Experiment Template for Evaluating the Understandability of Model Transformation Languages

Max E. Kramer
Karlsruhe Institute of Technology
max.e.kramer@kit.edu

Georg Hinkel
FZI – Research Center for Information Technology
hinkel@fzi.de

Heiko Klare
Karlsruhe Institute of Technology
heiko.klare@kit.edu

Michael Langhammer
Karlsruhe Institute of Technology
langhammer@kit.edu

Erik Burger
Karlsruhe Institute of Technology
burger@kit.edu

ABSTRACT

Several research approaches in the field of Model-Driven Engineering (MDE) are concerned with the development of model transformation languages. No controlled experiments have, however, been conducted yet to evaluate whether it is easier to write model transformations in a model transformation language (MTL) than in a general purpose programming language (GPPL). Such experiments are difficult to design and conduct. To write and maintain code in an MTL, it is necessary to understand the code. Thus, an evaluation of the effect on program comprehension is a first step towards empirically evaluating the benefit of model transformation languages.

In this study design paper, we propose an experiment template for empirically measuring the potential understandability gain of using an MTL instead of a GPPL. We discuss a randomized experiment setup, in which subjects fill out a paper-based questionnaire to prove their ability to understand the effect of transformation code snippets, which are either written with an MTL or a GPPL. To evaluate the influence of the language on the quality and speed of program comprehension, we propose two statistical tests, which compare the average number of correct answers and the average time spent.

Keywords

Model-Driven Engineering, Transformations, Model Transformation Language, Controlled Experiment, Program Comprehension

1. INTRODUCTION AND MOTIVATION

Model transformations are central to Model-Driven Engineering (MDE). Sometimes they are even called its "heart and soul" [SK03]. Thus, they should be supported by dedicated Model Transformation Languages (MTLs) that are designed to ease the development and maintenance of software that transforms models. For example, MTLs provide facilities to load and persist models and transformation traces. They also provide special language constructs to navigate in, create, or map model elements. MTLs have been developed because these recurring problems are not supported sufficiently by general purpose programming languages (GPPLs). To the best of our knowledge, there is however *no* literature on controlled experiments on the benefit of MTLs over GPPLs. MTLs have only been validated in case studies, where their concepts proved beneficial.

The model transformation community has not found a common notion of quality for model transformations yet [AB11]. This is a potential reason for the lack of empirical research on MTLs. One may be misled to think that the quality of a transformation *language* cannot be evaluated until we know how to evaluate the quality of a single *transformation*. This is, however, not true: We suggest a way to draw conclusions about an aspect of relative quality of two languages by comparing two transformations without a notion of absolute quality for an individual transformation or language.

In order to write or maintain code in a particular language, one has to be able to understand programs written in that language. Therefore, we call for evaluations of whether MTLs have a positive effect on program comprehension. They would be a first step towards an empirical evaluation of the question whether MTLs ease the development or maintenance of transformation code. Such an evaluation could answer the questions whether subjects are able to make more correct statements about what a model transformation does, or whether they require less time to do so when using an MTL. An MTL can only ease transformation development if it actually improves program comprehension: If a language is harder to understand than another language, there is little hope that it will be easier to develop transformations with it. Therefore, we suggest evaluating the understandability of MTLs. Many developers are more familiar with a GPPL than with an MTL. Thus, an observed improvement in comprehension that overcompensates this lack of experience with MTLs would be especially convincing.

In this study design paper, we propose a template or meta-protocol for an experiment that empirically measures understandability of MTL code compared to functionally-equivalent GPPL code. The goal is to obtain a reliable indicator for a lower bound of the potential benefit of using a language for development and maintenance of model transformations. We discuss an experiment setup, in which two randomly created subject groups obtain two snippets of transformation code developed for different parts of a requirements document. One group first obtains MTL code, and the other group first obtains GPPL code, but both groups also obtain code of the other language afterwards. For both languages, the subjects are asked to fill a questionnaire with false and valid statements regarding the functionality of the provided code. With this experiment, we want to assess whether the chosen language (independent variable) has an effect on the quality and speed of program comprehension. We measure the subjects' ability to correctly assess what the transformation code does, and the time required for this assessment (dependent variables, see [Woh+12]). To this end, we propose statistical testing of paired differences between the numbers of correct answers. If we observe a statistically significant benefit on correctness for the MTL, we also perform the test for the time spent on average for a correct answer. Both the gain of correctness and answering time are two potential indicators for understandability, which is complex to measure. Since we do not suggest analysing the results for both transformation parts separately, the proposed experiment setup is a *within-participants*, not a *between-participants* design.

We do discuss some specifics of MTLs in this paper. The proposed experiment setup is, however, not specific for model transformations. It is rather an experiment template, which can be instantiated to evaluate the influence on program comprehension for any programming language . It has not yet been shown for *any* transformation language that it improves program comprehension. Therefore, this paper is about comparing arbitrary transformation languages with arbitrary GPPLs. The proposed experiment can be used to compare two specific transformation languages, to evaluate which one is better suited for developing transformations. Such a comparison is, however, only reasonable after a positive influence on program comprehension was observed for an MTL when compared to a GPPL.

The remainder of this paper is structured as follows: In the next section, we formulate the research questions and hypotheses. In section 3, we present the plan for experiment preparation, execution, and analysis. Then, we discuss threats to validity in section 4. Afterwards, we instantiate our template with using a transformation scenario from the automotive domain in section 5. Finally, we discuss related work in section 6 and conclude the paper in section 7.

2. QUESTION AND HYPOTHESES

The quality of programming languages can be estimated by the ability of developers to assess what the code does and the time needed. This is also true for MTLs when compared to GPPLs or other MTLs. To measure this influence for a given language, we propose a controlled experiment. The research question is: Does the usage of an MTL T improve code comprehension in terms of correctness and speed when compared to a GPPL G?

The hypotheses concern a set of questions about the functionality of two functionally equivalent transformations written with T and G. For the two code comprehension dimensions of correctness and speed, the null hypothesis and alternative hypothesis are:

H_0^c**:** Usage of the language T has *no effect* on the number of *correctly* answered questions *or decreases* it when compared to G.

H_0^t**:** Usage of the language T has *no effect* on the average *time* needed to correctly answer a question *or increases* it when compared to G.

H_A^c**:** Usage of the language T *increases* the number of *correctly* answered questions when compared to G.

H_A^t**:** Usage of the language T *decreases* the average *time* needed to correctly answer a question when compared to G.

3. EXPERIMENT SETUP

The proposed experiment is divided into five phases:

1. A *preparation* phase to create the transformation and the functionality questionnaire.
2. A *training* phase to instruct the subjects.
3. A *warm-up* that is also used to obtain more data on their language-independent skills.
4. A phase for *conducting* the main experiment.
5. An *analysis* phase to perform statistical tests.

These phases, their individual activities, and the data flow are also shown in Figure 1.

3.1 Experiment Preparation

To prepare the experiment, we have to develop transformations with both languages, select the part to be used in the experiment, and create questions about its functionality.

Transformation Development

Before the experiment can be conducted, we need two model transformations that are functionally equivalent. One has to be written in an MTL T and the other in a GPPL G.

In order to increase the sensitivity, it is important to develop two transformations that *only* differ in terms of the used language and share as many properties as possible. For example, the coding style should be controlled for both transformations. This includes rules for variable names or comments, which are easier to define if the languages are similar, and harder for GPPLs and MTLs that have little concepts and constructs in common. To reduce sequencing effects, two groups of developers can develop both transformations in two phases using a counterbalanced setup: In the first phase, both groups write a transformation for each requirement half, but they switch languages between the halves and start with different languages and different requirement halves as shown in Figure 2. In the second phase, both groups improve the transformations of the other group so that they switch languages and halves again. We expect stronger sequencing effects when developers consecutively work with the same language than when working on the same requirements in succession. Therefore, we propose to let the groups switch the languages, and not the requirement halves, between the last action of the first phase and the first action of the second phase. We suggest improving the code in the second phase to gain a less heterogeneous transformation when combining both transformations of one language for both requirement halves. The resulting combination will, however, not be completely homogeneous. The counterbalanced setup ensures that both groups use both languages. Furthermore, it ensures that every line of code was written by one group and improved by the other group. Therefore, the different groups or requirement halves should have a similar impact on the transformations for both languages. Finally, we suggest that only developers that were not directly involved in the development of the transformation language T should write these transformations. The developers have to be trained to use the new language, but they should not know every twist of it. Such an extensive training would bear the risk that they produce transformation code that would not be written by average transformation developers, which are more likely to be experts in a GPPL than in a particular MTL. It is, however, difficult to find the right amount of training so that the results are neither biased because the training was too long nor unrealistic because it was too short.

Selection of a Transformation Part

For the experiment, a part of the transformation has to be selected, so that the subjects answer questions about its functionality. It is necessary to select only a part of the transformation in order to have a realistic setting. The subjects should read neither too few nor too much code. It should be readable in a limited amount of time in order to answer the questions. We also have to decide whether the presented code snippets should be self-contained or not. The metamodels of the models and the requirements for the transformation determine whether such self-contained parts exist, and how easily they can be found. To select a self-contained part of a transformation that is realistic *and* reasonably sized, it is one possibility to select representatives for groups of metaclasses that are similarly treated in the transformation. This has been successfully applied in several cases of the Transformation Tool Contest [Hor13; KPL15]. This

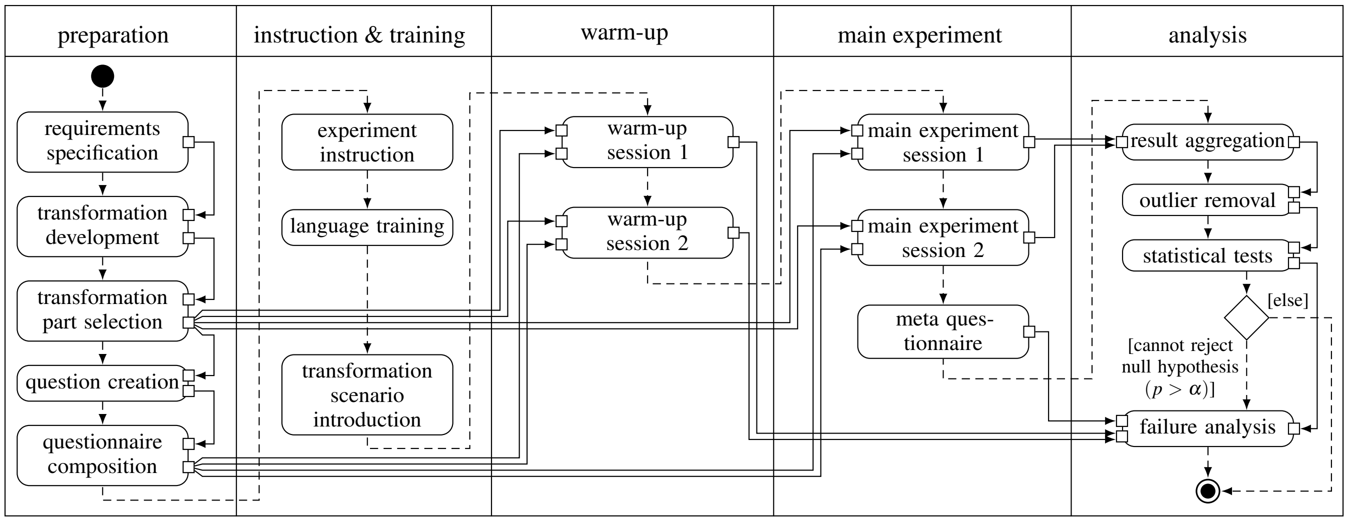

Figure 1: Experiment overview with activities, control flow ($\dashrightarrow$) and data flow ($\longrightarrow$) for all five experiment phases

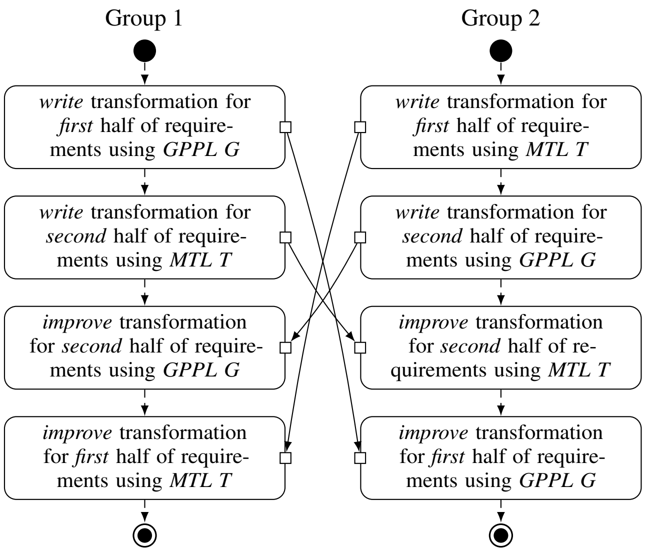
Group 1
Group 2
write transformation for *first* half of requirements using *GPPL G*
write transformation for *second* half of requirements using *MTL T*
improve transformation for *second* half of requirements using *GPPL G*
improve transformation for *first* half of requirements using *MTL T*
write transformation for *first* half of requirements using *MTL T*
write transformation for *second* half of requirements using *GPPL G*
improve transformation for *second* half of requirements using *MTL T*
improve transformation for *first* half of requirements using *GPPL G*

Figure 2: Counterbalanced developer groups for *developing transformations* for both halves of the requirements

makes it easier for a developer to guess the transformation parts for the other metaclasses, even if the code is not included.

Question Creation

In a final preparation step, the questions about the functionality of the transformation have to be prepared for the questionnaires of the warm-up and of the main experiment. Similar to the development of the transformation itself, we also suggest to let developers create these questions who did not develop the transformation language. This avoids questions that are directly targeting potential benefits of the language. The goal should be to obtain questions that are relevant and representative for the transformation, regardless of the language in use. To achieve this goal, we can provide question templates and ask the developers to create questions for a broad range of model elements, and for explicit complexity levels, e.g. simple, medium, and complex. In order to create good question templates, one could first perform experiments in order to learn which kind of questions are in fact asked by developers when encountering model transformations in general or the two MTLs in special. Such experiments have been performed for a couple of languages and different software projects [SMD06], but we expect only some of these questions to also apply to MTLs and model transformation software. An alternative would be to let developers create questions based on the requirements, instead of the transformation. Transformation requirements, however, only specify *what* should be transformed, and always leave some degrees of freedom with respect to *how* a transformation is realized. Therefore, such requirement-based questions would have the disadvantage that the questions cannot take into account how this freedom was used in a transformation.

The questions should only cover the previously chosen self-contained part of the transformation. Since only this part shall be given to the subjects, we have to hide all other code. Furthermore, we want to make sure that all questions assess whether subjects understand the transformation's functionality, and not whether they understand the question. Thus, we ask the developers to always create questions with a single correct answer and exactly three wrong answers (distractors). In order to also test higher levels of cognition, we will provide guidelines for writing such questions [HDR02].

Furthermore, we can also counterbalance the question preparation step, in a way similar to the transformation development step, using the same partition that was already used for the requirements. The goal of this counterbalancing is to obtain many different questions by always looking at both implementations. This also reduces the risk of language-specific questions. In contrast to the transformation development, we suggest, however, to use a complete inter-subject counterbalanced setup. According to this setup, every developer creates questions for every line of transformation code. All groups experience both the switching from T to G, and from G to T. A quarter of the developers (group 1, Figure 3) first create questions for one half of the transformation written in G and then for the other half written in T. Then they go back to the first half of the transformation, but stay with language T. Finally, they process the second half of the transformation using the first language G. This way, we try to ensure that developers create questions that are not concerned too much with language-specific realizations of a transformation, but with the language-independent functionality, which they see twice. We expect stronger sequencing effects when developers consecutively create

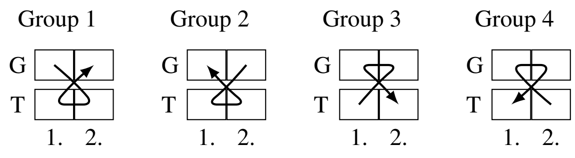

Figure 3: Counterbalanced developer groups for *creating questions* for both halves of the transformation subset

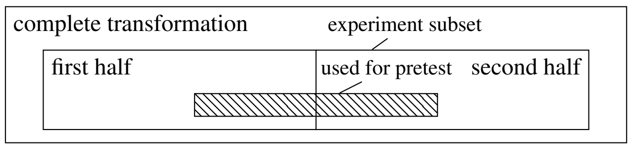

Figure 4: Self-contained transformation subset for the experiment and partition into halves for counterbalanced warm-up and main experiment

questions for the same functionality, than when they create questions for the same language in succession. Therefore, we propose to let the groups always switch the requirement halves.

Questionnaire Composition

Once we have obtained all questions, we have to select those that should be used in the questionnaires. This can be done, for example, by only selecting questions that relate to the same or similar functionality as another question by another developer. Alternatively, we can discard a fixed portion of questions for each level and developer based on a peer-review. In order to avoid the same bias towards language-related questions, we have to make this selection based on pre-defined criteria. Finally, we have to select a part of both transformation halves for the warm-up. We have to identify the questions that relate to these warm-up parts in order to produce the questionnaires. Both of the warm-up questionnaires and both of the main questionnaires should have exactly the same number of questions. The warm-up questionnaire should have about half of the size of the main questionnaire. On the one hand, the number of questions should not be too low, so that we create a challenging setting in which subjects do not use more time than they need on each question. On the other hand, it should not be too high so that subjects do not feel overwhelmed or demotivated. An estimate for a good number of questions for the warm-up and main experiment should be obtained by performing a pretest. This pretest should also be used to search for problems with the experiment setup and the developed transformations. Figure 4 depicts how a self-contained subset of the complete transformation has to be selected, as described in the previous section, and how this subset is divided into two halves, with parts for the warm-up and for the main experiment.

3.2 Instructions and Training

At the beginning of the experiment, the subjects receive instructions for the experiment setup, a training for both languages, and explanations of the transformation scenario. The instructions, training, and explanations should either be pre-recorded videos or be transmitted in written form to avoid any inaccuracy and bias.

First, the subjects are informed about the experiment setup, the division into two groups, and the repetition of the same task with different languages and different transformation parts. They learn that one group starts with language T and the other with language G. They are informed that both groups switch languages three times: during the warm-up, between the warm-up and the main experiment,

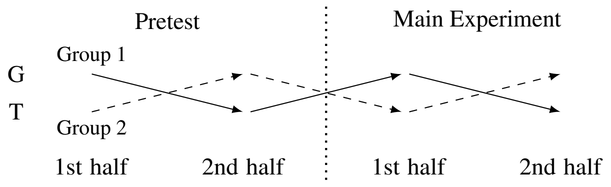

Figure 5: Counterbalanced subject groups answering questions during warm-up and main experiment

and during the main experiment. This setup is also depicted with different arrows for both groups in Figure 5. The subjects are also informed that the primary goal is to give as much correct answers as possible, and that the secondary goal is to do this as fast as possible.

Then, two short training sessions are performed to explain both languages. To avoid bias, both languages have to be presented with the same amount of detail, using similarly structured material, and with the same excitement of the instructor. Even if many subjects are familiar with the GPPL, it should still be treated equally as far as possible. Nevertheless, previous experiences with the GPPL may influence the experiment like in real development projects, where well-known GPPLs compete with less common MTLs. To avoid a sequencing effect, half of the subjects of both groups first receive a training for the GPPL, and the other half begins with the MTL.

Finally, the subjects are introduced to the domains of the models that are transformed, and the reason for the transformation. The subjects have to know what kind of model elements they can expect, what their purpose is, and why the models are transformed. They should not be informed how the transformation is meant to map elements to each other, or how the elements can be related. We want to assess whether the subjects are able to infer this from the code that they will receive during the warm-up and main experiment.

3.3 Warm-up

The warm-up has two goals: First, we let the subjects adjust to the new setting and to the task of filling out a questionnaire for transformation code snippets. Second, we want to obtain more data on the subjects' skills for *both* languages to analyze whether the randomized assignment to groups had an influence on the obtained results. This data can be used to analyze possible reasons if the statistical tests do not yield results that let us reject our null hypotheses.

For the first goal, it is important that the task performed during the warm-up is *identical* to the task of the main experiment. As we obtain questions for the complete transformation subset prior to identifying a part for both halves, we expect that the questions for the warm-up and for the main experiment fulfill this requirement. Therefore, we expect that the familiarization with the way the questions are formulated and with the overall transformation scenario and metamodels is mostly completed after the warm-up.

The second goal could also be reached with intra-subject counterbalancing: if every subject answered questions for both transformation halves for both languages (e.g., T_1, G_2, G_1, T_2), we could compare the individual differences in a better way. As the questions are the same for both languages, we might, however, observe a strong sequencing effect. Furthermore, the subjects would spend more time on the experiment, which may lead to negative maturation effects.

3.4 Main Experiment

The warm-up and the main experiment are conducted in the same way: In each of the four sessions, the subjects receive diagrams showing both complete metamodels, printouts of the transformation code with highlighted syntax, and questionnaires where all questions have one correct and three wrong answers. The subjects obtain 10 minutes for each warm-up session, and 20 minutes for each main session. They are asked to indicate when they answered all questions of one session, so that the instructor takes the questionnaire, and takes note of the time. Furthermore, subjects obtain the possibility to recover between the training, warm-up, and main experiment, as the next material is only handed out after a short break.

After the experiment, the subjects receive a short post-experiment questionnaire. With it, they can indicate whether they had problems with the task, felt stressed or distracted, and what evaluation goal they supposed. If we cannot reject our null hypotheses, this information from the post-experiment questionnaire can be used in addition to the warm-up results to obtain indicators on possible reasons, such as subject-expectancy effects.

We suggest to let the subjects read the transformation code on paper and not using an Integrated Development Environment (IDE). In this way, we try to avoid measuring noise that may be caused by different abilities to exploit features of a particular IDE, or different tool maturity. As a result, the proposed experiment cannot provide any evidence for potential benefits on program comprehension when an IDE is used, which may be different in terms of benefits than reading transformation code printouts. If our experiment showed that MTLs improve code comprehension on paper, it would just be a matter of engineering to develop tool support for the MTLs that does not ruin this benefit. It is, however, unlikely that such GPPL-equivalent tool support would be able to reverse a negative effect if our experiment showed that MTLs do not even result in better code comprehension on paper.

If our experiment showed that MTLs improve code comprehension using printouts, the experiment could be repeated using an IDE. If the result of the repeated experiment was that the advantage of the MTL is worse than with the printouts, or that users suddenly perform better using the GPPL, it could be the lack of tool support for the MTL. In this case it is, again, a matter of engineering to improve the tool support.

If we used a single IDE instead of paper in the proposed experiment, it is questionable whether the experiment would be more realistic, because developers usually are most productive when they use a particular IDE that they are most familiar with. Therefore, an experiment in which some subjects are familiar with the IDE would also evaluate whether these subjects have different problems with the IDE when using the different languages. An alternative could be to only have subjects who are used to a particular IDE. The results of such an experiment could, however, not be generalized to other IDEs and other subjects. An alternative experiment with a representative selection of IDEs that all provide exactly the same functionality for both the GPPL and the MTL is simply infeasible.

3.5 Analysis

The analysis of the results is performed in three steps: first, aggregated results are calculated, then outliers are treated, and finally statistical tests are performed.

Result Aggregation

In the first analysis step, aggregated results are calculated based on the raw data of answers chosen by the subjects, and time needed for it. For each of the four sessions of a subject, the number of correctly answered questions and the average time per correct answer are calculated. This average time per correct answer is calculated by dividing the time needed to answer all questions by the number of correct answers. Therefore, it is also influenced by the time needed to incorrectly answer questions. In a tool-based setting, we could instead calculate the time needed to correctly answer a question by measuring the times for individual questions and dividing only those times for correct answers by the number of correct answers. This is, however, not possible in a pen-and-paper-based setting, where we only record the overall time needed for all questions. For the languages G, T and the warm-up w or main experiment m and for each subject s, this yields four absolute numbers of *correct* answers $c_s^w(G)$, $c_s^w(T)$, $c_s^m(G)$, $c_s^m(T)$ and four average *times* per correct answer in seconds $t_s^w(G)$, $t_s^w(T)$, $t_s^m(G)$, $t_s^m(T)$. Only the values of $c_s^m(G)$ and $c_s^m(T)$ respectively $t_s^m(G)$ and $t_s^m(T)$ are used for statistical testing. Next, we calculate as intermediate results for each subject s the difference between the number of correct answers and the difference in time for the languages T and G as

$$\Delta c_s := c_s^m(T) - c_s^m(G) \qquad \Delta t_s := t_s^m(T) - t_s^m(G)$$

(We are interested to see which Δc_s are positive because s had more correct answers for T and which Δt_s are negative because s required less time per correct answer for T.) These values are only used to eliminate outliers that exhibit extraordinarily big differences between the languages as discussed below and to calculate two final informative results: For the set of all subjects S we calculate the average differences in correctness and time as

$$\Delta c := \frac{\sum_{s \in S} \Delta c_s}{|S|} \qquad \Delta t := \frac{\sum_{s \in S} \Delta t_s}{|S|}$$

Outlier Removal

In the second analysis step, outliers are treated based on the interquartile range. Subjects for which the correctness difference Δc_s, the time difference Δt_s or both measures are further away from the median than 1.5 times the interquartile range, are considered outliers (Tukey's test). These subjects are excluded from further analyses, and the average differences in correctness and time are recalculated without them. Winsorization of outliers by replacing their values with the nearest non-outlier is also possible. This would, however, have no effect on the proposed statistical testing, as we suggest the usage of the Wilcoxon test based on rank-sums that are not changed by winsorizing.

In contrast to common topics, such as object-oriented programming, experiences with Model-Driven Engineering and transformation development are not as widespread, and may be very different for individual developers. Thus, we suggest the removal of difference-based outliers, and the analysis of individual differences, instead of comparing the average correctness and time for language T and G. It would also be possible to remove outliers for subjects s with a very high or low number of correct answers c_s^m, and subjects with a very high or low time per correct answer t_s^m, regardless of the individual differences between the values for both languages (Δc_s or Δt_s). This would bear the risk that we accidentally remove measurements of subjects that were just very good or bad in reading code in comparison to the other subjects. Our difference-based outlier removal, however, only removes measurements of subjects that were very good in reading code of one language, but very bad in reading code of the other language. Such extraordinary differences should not be observed, even if T is much easier or harder to understand than G. Therefore, they should have an influence on our decision to keep or discard our null-hypothesis. A possible explanation for such extraordinary differences could be that a subject misunderstood a central language construct.

Statistical Testing

In the last analysis step, statistical tests are performed to see whether we can reject the null hypotheses and accept the alternative hypotheses. We cannot assume a normal distribution for the individual differences in correctness and time. Therefore, we assume an unknown distribution for both populations. We suggest to apply a one-sided Wilcoxon signed-rank test to the samples of individual correctness $c_s^m(G)$ and $c_s^m(T)$ for both languages with the significance level $\alpha = 0.05$, as it is nonparameteric. This test orders the individual absolute differences between the correctness for G and T for all subjects, and calculates two separate sums of the ranks R_s for positive and negative differences. The rank-sum W is used as test-statistic.

$$W = \sum_{s \in S} sgn(\Delta c_s) \cdot R_s$$

Under hypothesis H_0^c, the test statistic W follows a known distribution and therefore yields a probability p^c for observing sample distributions as collected.

We argue that a binomial test, which counts how often participants answered more correct answers using T, is not suitable in this case. This test tries to reject the hypothesis that the probability that more questions are answered correctly using T is at most 50%. Such a result would be valid from a mathematical point of view, but it ignores the scale of differences in correctly answered questions. A Student t-Test has the problem that it requires a normal distribution, an assumption that often does not hold. One can check this assumption with a quartile plot. If the assumption of a normal distribution does not hold, the Student t-Test is only an approximation. Textbooks on statistical tests often suggest at least 30 responses which is often not reached for experiments with subjects trained in MDE.

The Wilcoxon signed-rank does take these differences into account and is non-parametric, i.e. does not assume a particular assumption. Therefore, it is a good compromise between expressiveness and underlying assumptions.

If the p^c obtained from the Wilcoxon test is smaller than our significance level $\alpha = 0.05$, then we reject the null hypothesis H_0^c that the language T has no effect on the correctness of functionality and accept the alternative hypothesis H_A^c that T increases it. If p^c is even smaller than $\frac{\alpha}{2} = 0.025$, then we also perform the same statistical test for the paired differences between the times $t_s^m(G)$ and $t_s^m(T)$ to obtain p^t to see whether we can also reject the null hypothesis H_0^t . This second test is only performed for a sufficiently small p^c, because we have to control the familywise error rate when performing multiple hypothesis tests on the same set of data. In this case we suggest to apply the Holm–Bonferroni method to control the probability of making a Type I error. For two tests this method is equivalent to dividing the significance level by two.

If the first null hypothesis H_0^c cannot be rejected for the resulting p^t-value and our significance level $\alpha = 0.05$, then we suggest to analyze this. In such a case, the data from the warm-up and post-experiment questionnaire should be used in addition to the data of the main experiment. They should be analyzed to check whether the results are good but simply show that the MTL has no benefit or whether problems occurred. It would also be possible to see whether the null hypothesis can be rejected, if we assume that the warm-up was not necessary, and treat the warm-up sessions similar to the main experiment sessions in order to obtain additional data points. For this, we would first have to normalize the correctness for each language and subject during the warm-up and the main experiment. Both sums of the two proportions of correctly answered questions for each language could then be divided by two to obtain an average proportion of correct answers for each language. A one-sided Wilcoxon signed rank test could then be performed on the pairs of average proportions to test the original null hypothesis. Instead of independently performing outlier removal and adjusting the significance level for multiple tests, we could also use Wilcoxon's extension of the Yuen-Welch method [Wil12, pp. 317] as recently suggested by Kitchenham et al. [Kit+16].

3.6 Open Design Questions

We have already discussed the questions whether we should aim for a self-contained transformation part and how we should select questions for the questionnaires. In this section we present four more open questions of our study design.

1. Should we have an incentive for good performance for the subjects, e.g. by awarding a first prize to the subject with the highest number of correct answers and a second prize to the subject with the highest number of correct answers per minute?
2. Should we perform a lab experiment in which the training is performed live using a presentation, or should we invite every subject to conduct the experiment individually with written training text?
3. Should we split the transformation into quarters instead of halves in order to perform intra-subject counterbalancing T_1, G_2, G_3, T_4, perhaps even without a warm-up?
4. Should we winsorize outliers instead of truncating them, e.g. by replacing their Δc_s or Δt_s with the nearest non-outlier?

4. THREATS TO VALIDITY

We will briefly discuss threats to internal and external validity of the proposed experiment, and discuss how they can be addressed. These are threats to construct validity. Content validity has two facets for the template and its instantiation: Is the gain of correctness or speed in answering questions in general a reliable indicator for the quality of a language? Are the created questions for an instance of the template really evaluating whether a subject understands the developed transformations?

4.1 Internal Validity

We have identified threats of history (external influences between test sessions), maturation (internal changes during sessions), and sequencing effects (from session to session). We address this by only performing a short experiment that requires $20+5+20+5+40 = 90$ minutes for the preparation, warm-up, and main experiment. We include two breaks of 5 minutes. To avoid instrumentation effects, we do not perform our experiments using an IDE, but on pen-and-paper basis. We measure the required time that is indicated by the subjects. To circumvent selection and mortality effects, we randomly assign subjects to the two groups that either start with language T or with language G, and let every subject answer questions for both languages. To reduce the risk of a subject-expectancy effect, we explain both languages in the same way. We do not disclose that the control language G is used as a baseline for evaluating potential benefits of our treatment with language T. Since the MTL T is less common than the GPPL G, subjects may still be able to guess that potential benefits of T should be evaluated. Therefore, we can only try to lower the risk that subjects try to perform better when answering questions about T, but we cannot completely eliminate it. Furthermore, there will be a risk of intratreatment interactions because different subjects may exhibit different self-portrayal. To avoid an observer-expectancy effect, the transformations, and questions are created by developers that did not develop the evaluated language T. Furthermore, all subjects did not take part in the design of the experiment. The training is also performed by an instructor that developed neither the language nor the experiment. Therefore, the experimenters have only little influence on the transformation development, question development, training, and execution of the experiment. Thus, it would be no problem if the experimenters already tend to consider it more likely that the null hypothesis will be rejected or that it will not be rejected. Furthermore, there is no degree of freedom when performing the statistical tests no degree of freedom remains. Therefore, we do not expect such a potential tendency to have an influence on the results of the experiment.

4.2 External Validity

The threats to external validity of the proposed study are mainly aptitude-treatment interactions, situational specifics, and reactivity effects. Aptitude-treatment interactions, which are also called population effects, can have a positive or negative influence on the experiment. The subject sample could be positively biased because subjects of the experiment may have a greater ability to understand a newly presented transformation language than average developers. This could apply in an academic experiment context if subjects are, for example, computer science students, but also if the experiment is conducted with professional software developers that were exposed to more different languages than average developers, or that have a higher motivation for learning new languages. Such a subject sample bias could lead to the observation of a stronger influence of the language than would be observed in real software development projects. A negative population effect could result from the short training, which is not realistic: Industrial developers usually have more than 20 minutes to learn a new language before they are asked to read and understand code written with it. We have discarded an alternative setup where subjects obtain training material for self-study at home prior to the experiment. This could lead to an increased variance due to different motivation for reading training material in advance. Such a variance in terms of skills could decrease the sensitivity, which is the probability not to miss the chance to falsely keep the alternative hypothesis. Situational effects could be observed if developers performed very different during the everyday usage of an IDE, since they offer navigation and documentation facilities in contrast to the pen-and-paper scenario of our experiment. We are, however, convinced that the risk to obtain no significant data due to different usage of IDE features is bigger than the threat to external validity of not using an IDE. The IDE support is missing for both languages, and we do not see a reason why one language should benefit much more from IDE features in terms of program comprehension than another language, if such features exist for both languages and are similarly realized. Furthermore, we could observe reactivity effects because the subjects know that they take part in a scientific experiment, and therefore could be more effective in learning the new language than in a usual everyday setting. Finally, our transformation scenario can be seen as a sample for model transformations in general, which can be biased, e.g. because we chose transformation requirements for which our MTL has a stronger influence than on average transformations. It is hard to estimate whether the scenario and the obtained transformation are representative because there is little research on representative model transformations.

5. EXPERIMENT INSTANTIATION

We plan to instantiate the general experiment template presented in this paper in order to evaluate a change-driven transformation language for multi-model consistency in comparison with Java. The subjects will be graduate students. For the transformation scenarios, we have two options: The first one is a transformation from software architecture models to source code. The second one is a transformation for software architecture models in the automotive domain. We already have transformations for both languages in the first transformation scenario. These transformations were, however, not developed using the counterbalanced design proposed of this paper. Therefore, we have decided not to use them for the evaluation, but to develop new transformations for the second transformation scenario from the automotive domain. It involves the metamodels ASCET and SysML, which are briefly explained later on.

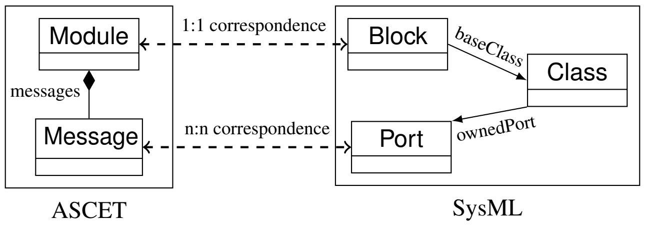

Figure 6: A simplified snippet from the automotive metamodels ASCET and SysML and the correspondence relation (dashed).

The MTL provides declarative and imperative constructs for restoring consistency between models of different modeling languages after monitored changes, and can neither be used for batch transformations, nor for synchronizing already existing models. It is named MIR for "mappings, invariants, and responses" and is still under development in an academic research project. It uses the language development framework Xtext [Eff+12], and is based on the expression language Xbase [Eff+12], which is also used in Xtend. Therefore, the MTL can also be seen as an extension to the Java language that provides specific features for change-driven inter-model consistency, such as predefined change triggers, transparent trace models, or simplified model element creation and removal.

The subjects of the experiment will be graduate computer science students that have participated in a practical course on model-driven software development, which we have offered four times during the last two years. They have been trained in meta-modeling and model transformations, but have not used the language to be evaluated.

The transformation requirements will specify how functional units of embedded software in the automotive domain and their compositions should be kept consistent with descriptions of the hardware. Figure 6 shows a simplified snippet of the automotive metamodels in use, ASCET and SysML, and their correspondence relation. For this transformation scenario and the depicted metaclasses, references, and correspondence relations, a possible question for the experiment questionnaire could be: "The transformation maps blocks to modules. How are related elements transformed?

a) When a port is added to a block, a message is added to the module that corresponds to the block.
b) When a message is deleted from a module, the port corresponding to the message is deleted from the block that corresponds to the module.
c) When a block is created, a module is created, and for each port that is owned by the base class of the block, a message is added to the module.
d) When a module is deleted, the corresponding block is deleted, and all ports that correspond to the messages of the module are also deleted."

6. RELATED WORK

To the best of our knowledge, there is no literature on a controlled experiment to empirically assure the advantages of any MTL over any other MTL or GPPL. There is a series of Transformation Tool Contests (TTCs), where contestants are asked to submit solutions to model transformation problems using the tool of their choice. This often offers a good comparison between the participating tools. However, only the results of few editions have been published [Var+07;

Kol+14; Ros+14]. These contests resemble realistic scenarios, since relatively complex transformation tasks are completed in several languages. The solutions are typically created by the tool authors, so that the contests compare the functionality that the tools offer, rather than what actual developers could do with them. Moreover, these contests typically do not attract solutions in GPPLs, such that no insights on a comparison to general purpose tools can be drawn.

There is, however, literature on controlled experiments that evaluated the influence of other Domain-Specific Languages (DSLs).

One experiment, for example, analyzed the influence of dynamic object process graphs on program understanding [Qua08]. Subjects solved feature location tasks in two interleaved experiments for two software systems. Four groups of subjects were used to analyze all possible sequences of task input and graph support or not (independent variable). For one of the two systems the alternative hypotheses that dynamic object process graphs result in faster comprehension with less errors could be rejected. With this experiment it was analyzed for two separated software systems whether a DSL that is integrated in an IDE helps subjects in locating certain program behavior. The experiment proposed in this paper, however, uses printed code for a single software system and lets subjects assess what a program does and not where it has a certain feature.

Another experiment was able to show a reduction of the total time spent and an improvement of correctness of answers to comprehension tasks when using trace visualization [CZD11]. In contrast to our experiment, it used IDEs and essay questions. Furthermore, it did not evaluate only the time needed per correct solution, but the time needed for false and correct answers.

A series of three other experiments compared the DSLs for feature diagrams, graph descriptions and graphical user interfaces with the usage of library code [KMC12]. During all three experiments undergraduate students answered questions in two experiment sessions for the DSL and GPPL after corresponding training sessions. In contrast to the experiments of the papers above and to the experiment proposed in this paper, subjects were not assigned to different groups that either started with the DSL or GPPL. Instead, every experiment was conducted twice. Once starting with the DSL and once starting with the GPPL. The authors also analyzed the efficiency in terms of correct answers per minute in addition to the percentage of correct answers and the overall time. A positive effect on the correctness and the amount of time needed was shown for all three DSLs.

7. CONCLUSIONS

In this paper, we have discussed a design template of a controlled experiment to evaluate whether the usage of a model transformation language T increases the ability to correctly assess the functionality of a given transformation code, or the time needed for such a correct assessment. This is done in comparison to a general purpose transformation language G. We have motivated the problem, presented a research question, formulated null and alternative hypotheses, and described our study design in detail. For a preparatory phase, we have described how to develop two functionally equivalent transformations, using both languages T and G. We have described how to create questions that test a subject's ability to understand the functionality of these transformations. We have also described how to conduct a warm-up and main experiment, in which subjects answer these questions for transformation code snippets of both languages. Then, we have described how to calculate results, and how to perform statistical hypothesis testing. We have also discussed open design questions and threats to validity. Finally, we presented an instantiation of the template for evaluating a change-driven MTL in comparison with Java and discussed related work.

References

[AB11] M. F. van Amstel and M. G. J. van den Brand. "Model Transformation Analysis: Staying Ahead of the Maintenance Nightmare." In: *Theory and Practice of Model Transformations*. Springer, 2011, pp. 108–122.

[CZD11] B. Cornelissen et al. "A Controlled Experiment for Program Comprehension through Trace Visualization." In: *IEEE Transactions on Software Engineering* 37.3 (2011), pp. 341–355.

[Eff+12] S. Efftinge et al. "Xbase: Implementing Domain-specific Languages for Java." In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*. GPCE '12. ACM, 2012, pp. 112–121.

[HDR02] T. M. Haladyna et al. "A Review of Multiple-Choice Item-Writing Guidelines for Classroom Assessment." In: *Applied Measurement in Education* 15.3 (2002), pp. 309–333.

[Hor13] T. Horn. "The TTC 2013 Flowgraphs Case." In: *Sixth Transformation Tool Contest (TTC 2013)*. EPTCS. 2013.

[Kit+16] B. Kitchenham et al. "Robust Statistical Methods for Empirical Software Engineering." In: *Empirical Software Engineering* (2016), pp. 1–52.

[KMC12] T. Kosar et al. "Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments." In: *Empirical Software Engineering* 17.3 (2012), pp. 276–304.

[Kol+14] S. Kolahdouz-Rahimi et al. "Evaluation of model transformation approaches for model refactoring." In: *Science of Computer Programming* 85 (2014), pp. 5–40.

[KPL15] G. Kulcsár et al. "Object-oriented Refactoring of Java Programs using Graph Transformation." In: *Proceedings of the 8th Transformation Tool Contest (TTC)*. 2015, pp. 53–82.

[Qua08] J. Quante. "Do Dynamic Object Process Graphs Support Program Understanding? - A Controlled Experiment." In: *16th IEEE International Conference on Program Comprehension*. 2008, pp. 73–82.

[Ros+14] L. M. Rose et al. "Graph and model transformation tools for model migration." In: *Software & Systems Modeling* 13.1 (2014), pp. 323–359.

[SK03] S. Sendall and W. Kozaczynski. *Model transformation the heart and soul of model-driven software development*. Tech. rep. 2003.

[SMD06] J. Sillito et al. "Questions Programmers Ask During Software Evolution Tasks." In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2006, pp. 23–34.

[Var+07] D. Varró et al. "Transformation of UML models to CSP: A case study for graph transformation tools." In: *International Symposium on Applications of Graph Transformations with Industrial Relevance*. Springer. 2007, pp. 540–565.

[Wil12] R. Wilcox. *Introduction to Robust Estimation and Hypothesis Testing*. Third. Academic Press, 2012.

[Woh+12] C. Wohlin et al. *Experimentation in Software Engineering*. Springer, 2012.